CHAPITRE 2

LES PROTOCOLES DE BASE

CONTENU

RESUME

Le but de ce chapitre est d'introduire quelques problèmes de calcul et solutions qui sont à la fois basique et primitifs. Ces problèmes sont basiques dont le sens où leurs solutions sont communément exigées pour le fonctionnement du système. Ils sont primitifs dans le sens où leurs solutions représentent des étapes ou modules de protocoles et calculs complexes (e.g., *parcours et construction d'une arborescence couvrante*).

Tous les problèmes considérés ici exigent dans leurs solutions la restriction de *Connectivity* (CN) (i.e., chaque entité est atteignable à partir de n'importe quelle autre entité). En général et sauf indication contraire, nous supposons aussi la *fiabilité totale* (TR) et des *liaisons bidirectionnelles* (BL). Ces trois restrictions sont communément utilisées ensemble et l'ensemble **R**= {BL, CN, TR} sera appelé ensemble des restrictions standards.

2.1 LA DIFFUSION

2.1.1 Problème

Considérons un système distribué où seule une entité, *x*, connaît une information importante, et souhaite partager cette information avec toutes les autres entités dans le système (voir figure 2.1). Ce problème s'appelle la *Diffusion* (**Bcast**).

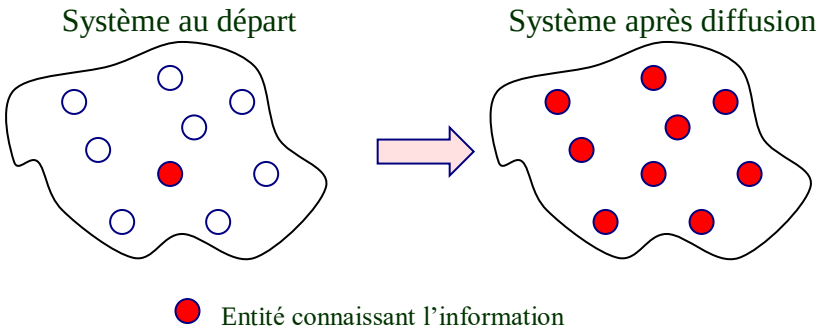

FIGURE 2.1: Processus de diffusion

Pour résoudre ce problème il faut concevoir un ensemble de règles qui, une fois exécuté par les entités, conduit (au bout d'un temps fini) à une configuration où toutes les entités connaissent l'information. La solution doit fonctionner quelque soit l'entité qui possède l'information au départ. De la définition même du problème, il y a une supposition, *Initiateur Unique* (UI), que seule l'entité possédant l'information lance l'exécution du protocole. Réellement, cette supposition est encore restreinte, car l'initiateur unique doit être celui possédant initialement l'information. Nous noterons cette restriction par UI+.

Pour résoudre ce problème, chaque entité sera impliquée. Par conséquent, la diffusion nécessite la connectivité (CN) sinon certaines entités ne pourront jamais recevoir l'information. Nous avons déjà exposer une solution (protocole d'inondation) à ce problème sous deux restrictions additionnelles : Fiabilité totale (TR) et liaisons bidirectionnelle (BL). Nous avons donc pour ce problème l'ensemble standards de restrictions **R** = {BL, CN, TR}.

2.1.2 Coût de la diffusion

Comme vu en chapitre 1, le protocole inondation requiert $O(m)$ messages et, au pire des cas, $O(d)$ unité de temps idéal, où d est le diamètre du réseau. La première question naturelle est : ces coûts peuvent-ils être réduits significativement en utilisant une approche différente ou technique et si oui de combien ? Cette question est équivalente à celle-ci : Quelle est la complexité du problème de diffusion ? Pour répondre à cette question, nous devons établir une limite inférieure pour la recherche d'une limite f (typiquement, une fonction dépendant de la taille du réseau) et prouver que le coût de n'importe quel algorithme solution est au moins égal f. En d'autres termes, une limite inférieure est nécessaire sans tenir compte du protocole et dépend uniquement du problème. C'est une indication de la complexité réelle du problème.

Nous notons par $M(Bcast/RI+)$ et $T(Bcast/RI+)$ la complexité en messages et en temps de la diffusion sous les restrictions $RI+ = R \cup UI+$.

Une limite inférieure du total d'unités de temps idéal requis pour la diffusion est simple à trouver : Chaque entité doit recevoir l'information indépendamment de sa distance de l'initiateur et toute entité peut être initiatrice. Par conséquent, au pire des cas,

$$T(Bcast/RI+) \geq Max\{d(x, y) : x, y \in V\} = d. \quad (2.1)$$

Le fait que le protocole inondation réalise la diffusion en d unité de temps idéal signifie que la limite inférieure est atteignable et que le protocole inondation est *optimal en temps*. En d'autres termes, nous connaissant exactement la complexité en temps idéal de la diffusion.

Propriété 2.1.1 *La complexité en temps idéal de la diffusion sous RI+* est $\Theta(d)$.

Considérons la complexité en messages. Une limite inférieure triviale en nombre de message est facile à trouver : A la fin toute entité doit connaître l'information, ainsi un message doit être reçu par chacune des $n-1$ entités qui n'avait pas l'information initialement. Ainsi, $M(Bcast/RI+) \geq n - 1$.

Avec un petit effort supplémentaire, on peut déduire une limite inférieure plus précise.

Théorème 2.1.1 $M(Bcast/RI+) \geq m$.

Preuve. Supposons qu'il existe un protocole correcte A, qui à chaque exécution sous **RI**+ sur le graphe G, utilise moins que $m(G)$ messages. Ceci signifie qu'il y a au moins un arc dans G sur lequel aucun message n'est transmis dans aucune direction durant l'exécution de l'algorithme. Considérons une exécution de l'algorithme sur G, et soit $e=(x, y) \in E$ la liaison sur laquelle aucun message n'est transmis par A. Maintenant, construisons un nouveau graphe G' à partir de G en enlevant l'arc e, et en ajoutant un nouveau nœud z et deux arcs $e1 = (x, z)$ et $e2 = (y, z)$ (voir figure 2.2). Mettons z

dans un état de non initiateur et exécutons exactement *A* sur le nouveau graphe *G'*. Vu qu'aucun message ne passe sur l'arc (*x*, *y*) dans la première exécution, ni *x* ni *y* n'enverra un message à *z* dans l'exécution courante. Par conséquent, *z* ne recevra jamais l'information (i.e., changera de statut). Ceci est en contradiction avec le fait que *A* est un protocole de diffusion correct.

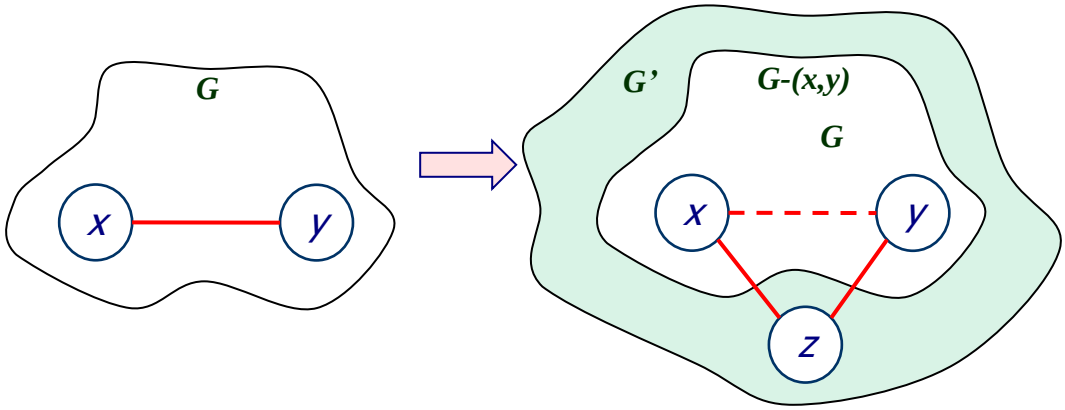

FIGURE 2.2: La diffusion nécessite un message sur chaque liaison

Ceci signifie que n'importe quel algorithme de diffusion requiert $\Omega(m)$ messages. Vu que le protocole inondation résout le problème avec 2m-n+1 messages, il s'avère que $M(Bcast/RI+) \leq 2m - n + 1$. Partant du fait que la limite inférieure et la limite supérieure sont de même ordre de grandeur, nous pouvons résumer :

Propriété 2.1.2 *La complexité de la diffusion sous RI+* est $\Theta(m)$.

La conséquence immédiate est, en ordre de grandeur, le protocole inondation est une solution optimale en messages. Ainsi, si on veut concevoir un nouveau protocole qui améliore le coût $2m - n + 1$, le mieux qu'on puisse faire est de réduire la constante 2. Dans tous les cas, à cause du Théorème 2.1.1, la réduction ne peut pas baisser la constante à moins de 1.

2.1.3 La diffusion dans les réseaux spéciaux

Les résultats obtenus jusque là s'applique à des solutions génériques, càd les solutions qui ne dépendent pas de G et peuvent ainsi être appliquées indépendamment de la topologie de communication (pourvu qu'il soit indirect et connecté). Dans ce qui suit, nous considérons la diffusion dans des réseaux spéciaux. Tout au long nous considérerons les restrictions standard et UI+.

Diffusion dans un arbre. Considérons le cas ou *G* est un arbre, càd, *G* est connecté et n'a aucun cycle. Dans un arbre $m = n-1$; donc l'utilisation du protocole d'inondation pour la diffusion dans un arbre coûtera $2m - (n - 1) = 2(n - 1) - (n - 1) = n - 1$ messages.

IMPORTANT. Ce coût est atteint même si les entités ne savent pas que le réseau est un arbre.

IMPORTANT. Comme effet secondaire intéressant de la diffusion dans un arbre est que l'arbre aura comme racine l'entité qui initie la diffusion.

Diffusion dans les hypercubes orientés. Une topologie de communication qui est communément utilisée comme réseau d'interconnexion est l'*hypercube étiqueté à K dimensions* noté H_k. Un hypercube orienté H_1 de dimension $k = 1$ est juste une paire de nœuds nommé (en binaire) "*0*" et "*1*," connecté par une liaison étiqueté par "*1*" aux deux noeuds. Un hypercube H_k de dimension $k > 1$ est obtenu à partir de deux hypercubes de dimension $k-1$ (H'_{k-1} et H''_{k-1}) en connectant les nœuds de même nom avec une liaison étiqueté par k aux deux nœuds. Le nom de chaque nœud dans H'_{k-1} (respectivement H''_{k-1}) est ensuite modifié en le préfixant par le bit 0 (respectivement, 1); voir figure 2.3.

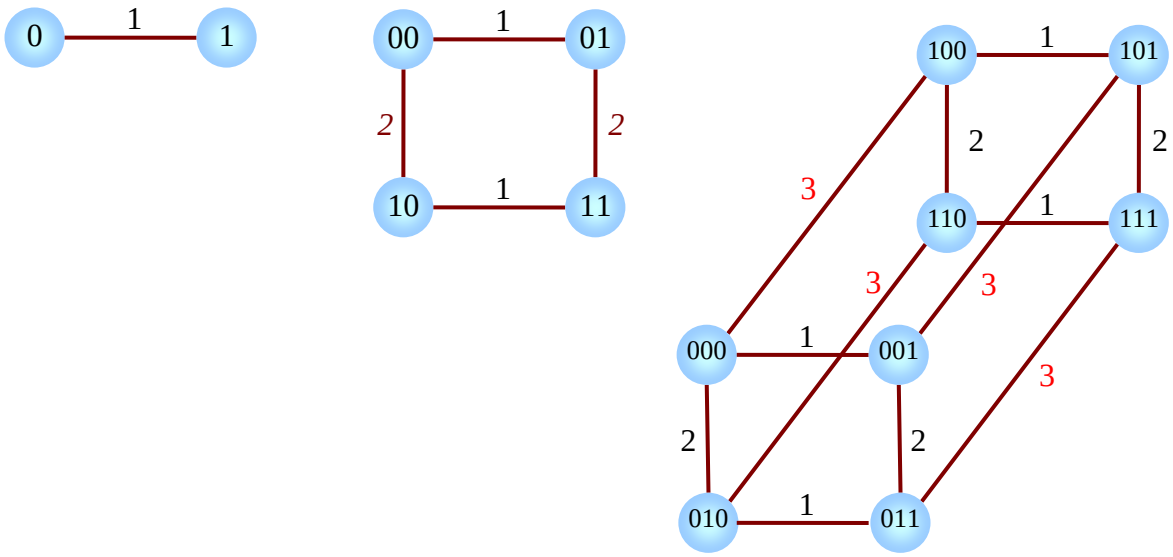

FIGURE 2.3: Réseaux Hypercubes orientés

Ainsi, par exemple, le nœud "*0010*" dans H'_4 sera connecté au nœud "*0010*" dans $H''4$ par un lien étiqueté $l=5$, et leurs noms deviennent "*00010*" et "*10010*," respectivement.

Cet étiquetage l des liaisons est symétrique (i.e., $l_x(x,y) = l_y(x,y)$) et est appelé étiquetage d'hypercube par dimension.

IMPORTANT. Ces noms sont utilisés uniquement à des fins de description; ils ne sont pas connus par les entités. Par contre, les étiquettes des liens (i.e., les numéros de ports) sont connues aux entités selon l'axiome de l'orientation locale.

Un hypercube de dimension k a $n = 2^k$ nœuds, chaque nœud a k liens, étiquetés *1, 2, . . . , k*. Donc, le nombre total de liens est $m = nk/2 = (1/2)\, n \log n = O(n \log n)$.

Une application directe de inondation dans un hypercube nécessite un nombre de messages égale à :

$$2m - (n-1) = n \log n - (n-1) = n \log n/2 + 1 = O(n \log n)$$

Cependant les hypercubes sont hautement structurés avec des propriétés intéressantes. Nous pouvons exploiter ces propriétés pour réaliser une diffusion plus efficace. Evidemment, on faisant cela, le protocole ne peut être utilisé dans d'autres réseaux. Considérons la stratégie suivante :

Stratégie *HyperFlood*:

1. L'entité initiatrice envoi un message à toutes ses voisines.

2. Un nœud qui reçoit un message d'un lien étiqueté l enverra des messages seulement aux voisins qui ont une étiquette $l' < l$.

Remarque. La seule différence entre *HyperFlood* et inondation normale réside dans l'étape 2: Au lieu d'envoyer le message à tous les voisins excepté l'émetteur, l'entité transmettra des messages à seulement une partie d'entre eux en fonction du label du port par lequel le message a été reçu.

Cette stratégie réalise la diffusion en utilisant seulement $n - 1$ messages (au lieu de $O(n \log n)$). Examinons d'abord la terminaison et la *correctness*.

Soit $H_k(x)$ le sous graphe de H_k induit par les liens où les messages sont envoyés par le protocole *HyperFlood* lorsque x est l'initiateur. Il est clair que chaque nœud dans $H_k(x)$ recevra l'information.

Lemme 2.1.1 *HyperFlood se termine correctement*.

Preuve. Soit x l'initiateur ; partant de x, les messages sont envoyés sur les liens avec des étiquettes décroissantes et si y reçoit le message d'un lien 4, il le transmettra uniquement sur les ports 1, 2, et 3. Pour prouver que chaque entité recevra l'information envoyée par x, nous avons besoin de montrer que, pour chaque nœud y, il y a un chemin de x à y tel que la séquence des étiquettes sur le chemin de x à y est décroissante (Notons que les étiquettes sur le chemin ne sont pas forcément des entiers consécutives). Pour cela, nous utiliserons la propriété des hypercubes suivante.

Propriété 2.1.3 *Dans un hypercube H_k de dimension k, tout nœud x est connecté aux autres nœuds par un chemin $\pi \in [x, y]$ tel que $\Lambda(\pi)$ est une séquence décroissante.*

Preuve. Considérons les noms à k-bits de x et y dans H_k: $<X_k, X_{k-1}, \ldots, X_1, X_0>$ et $<Y_k, Y_{k-1}, \ldots, Y_1, Y_0>$. Si $x \neq y$, ces deux chaînes diffèreront sur $t \geq 1$ positions.

Soit $j_1, j_2, \ldots, j_t$ les positions en ordre décroissant, càd, $j_i > j_{i+1}$. Considérons maintenant les nœuds $v_0, v_1, v_2, \ldots, v_t$, où $v_0 = x$, et le nom de v_i diffère de celui de v_{i+1} seulement au niveau de la position ji+1. Ainsi, il y a un lien étiqueté j_{i+1} connectant v_i to v_{i+1}, et clairement $v_t = y$. Mais cela signifie que $< v_0, v_1, v_2, \ldots, v_t >$ est un chemin de x à y, et la séquence des étiquettes sur ce chemin est $< j_1, j_2, \ldots, j_t >$, qui est décroissante. Ainsi, $H_k(x)$ est connecté et couvre (i.e., il contient tous les nœuds de) H_k, indépendamment de x. En d'autres termes, en un temps fini, chaque entité aura l'information.

Concentrons maintenant sur le coût de l'*HyperFlood*. Notons d'abord que

$$\boldsymbol{M}[HyperFlood/H_k] = n - 1. \quad (2.2)$$

Pour prouver que $n - 1$ messages seulement seront envoyés par la diffusion, nous avons juste besoin de montrer que chaque entité recevra le message une seule fois. Ceci est vrai car pour chaque x, $H_k(x)$ ne contient pas de cycle.

Ces coûts sont les meilleurs que tout algorithme de diffusion peut atteindre dans un hypercube indépendamment de connaissances supplémentaires qu'ils peuvent avoir. Cependant, ils sont obtenus ici sous la restriction supplémentaire que le réseau hypercube à *k dimension avec un étiquetage de dimension*; càd, sous $H = \{(G, \lambda) = H_k\}$. En résumé, nous avons :

Propriété 2.1.4 *La complexité en temps idéal de la diffusion dans un hypercube a k dimensions avec un étiquetage de dimension RI+ est* $\Theta(k)$.

Propriété 2.1.5 *La complexité en message de la diffusion dans un hypercube a k dimensions avec un étiquetage de dimension RI+ est* $\Theta(n)$.

IMPORTANT. Ce qui nous permet de dépasser la limite inférieure $\Omega(m)$ donnée par le théorème 2.1.1 est le fait que nous limitons l'applicabilité du protocole.

Diffusion dans les graphes complets. Parmi toutes les topologies, le graphe complet est celui ayant le plus de liens : Chaque entité est connectée à toutes les autres ; ainsi, $m = n(n - 1)/2 = O(n^2)$ (rappelons qu'il s'agit de liens bidirectionnels), et $d = 1$.

L'utilisation d'un protocole générique requiert $O(n^2)$ messages. Mais cela n'est pas réellement nécessaire. La diffusion dans un graphe complet est plus simple à réaliser : car chaque nœud est relié à tous les autres, l'initiateur n'aura qu'à envoyer l'information à tous ses voisins. (i.e., exécuter la commande "*send(I) to N(x)*") et la diffusion sera terminée. Ceci utilise seulement $n - 1$ messages et $d = 1$ temps idéal.

Il est clair que le protocole ainsi décrit, nommé *KBcast*, ne fonctionne que dans un graphe complet, càd sous la restriction supplémentaire $K \equiv$ "*G est un graphe complet*."

En résumé :

Propriété 2.1.6 *La complexité en messages et en temps idéal de la diffusion dans un graphe complet sous* RI+ sont *M(Bcast/RI+ ;K) = n − 1 et T(Bcast/RI+ ;K) = 1, respectivement*.

2.2 LE REVEIL

2.2.1 Le Réveil générique

Très souvent, dans un environnement distribué, on se retrouve face à la situation suivante : Une tâche doit être réalisée dans laquelle toutes les entités doivent être impliquées ; cependant, seules quelques unes sont actives de manière indépendante (à cause d'événements spontanés ou ayant fini un calcul précédant) et prêtes pour le calcul, les autres sont inactives et ne sont même pas au courant du calcul qui doit être effectué. Dans ces situations, pour réaliser la tâche, nous devons garantir que toutes les entités deviennent actives. Il est clair que cette étape préliminaire ne peut être effectuée que par les entités qui sont déjà actives; cependant, elles ne savent pas quelles autres entités (s'il y en a) sont déjà actives.

Ce problème est appelé *Réveil* (*Wake-Up*) : Une entité active est usuellement dite réveillée, une entité inactive est dite endormie. La tâche consiste à réveiller toutes les entités; voire figure 2.4.

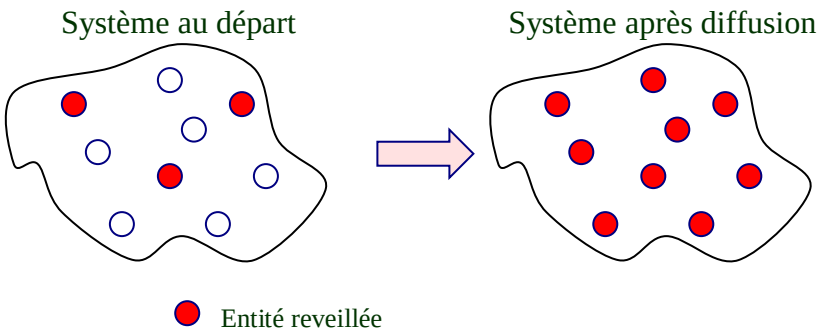

FIGURE 2.4: Processus de réveil

Il n'est pas difficile de voir la relation entre la diffusion et le réveil : La diffusion est un réveil avec une seule entité réveillée au départ ; inversement, le réveil est une diffusion avec éventuellement plusieurs initiateurs (i.e., initialement plus d'une entité possède l'information).

En d'autres termes, la diffusion est juste un cas spécial du problème de réveil. Il est intéressant, mais pas étonnant, de constater que la stratégie de inondation, utilisée pour la diffusion, résout réellement le problème le problème plus général du réveil. Le protocole modifié, appelé *WFlood*, est décrit en figure 2.5. Initialement toutes les

entités sont endormies ; n'importe laquelle peut se réveiller spontanément et lance le protocole.

Il n'est pas difficile de vérifier que le protocole se termine correctement sous les restrictions standard.

Concentrons nous sur le protocole *WFlood*. Le nombre de messages est au moins égal à celui de la diffusion ; en fait, il n'en est pas supérieur.

$$2m - n + 1 \leq M[WFlood] \leq 2m. \quad (2.4)$$

Comme la diffusion est un cas spécial du réveil, il n'y a pas d'améliorations (excepté peut être en terme d'une constante) :

$$M(Wake\text{-}Up/R) \geq M(Bcast/RI+) = \Omega(m)$$

Le temps idéal sera, en général, plus petit que celui de la diffusion :

$$T(Bcast/RI+) \geq T(Wake\text{-}Up/R)$$

```
PROTOCOL WFLOOD .
        • STATUS VALUES: S = {ASLEEP,AWAKE};
                         SINIT = {ASLEEP};
                         STERM = {AWAKE}.
        • RESTRICTIONS: R.
ASLEEP
  SPONTANEOUSLY
          BEGIN
            SEND(W) TO N(X);
            BECOME AWAKE;
          END
  RECEIVING(W)
          BEGIN
              SEND(W) TO N(X) − {SENDER};
              BECOME AWAKE;
          END
```

FIGURE 2.5: Réveil par inondation

Cependant, dans le cas d'un seul initiateur, les deux cas coïncident. Comme les limites supérieure et inférieure en ordre de grandeur, nous pouvons conclure que le protocole *WFlood* est, à la fois, optimal en *message* et optimal en temps dans le pire des cas. La complexité est résumée par les deux propriétés suivantes :

Propriété 2.2.1 *La complexité en message du réveil sous R est $\Theta(m)$.*

Propriété 2.2.2 *La complexité en temps idéal dans le pire des cas du réveil sous R est $\Theta(d)$.*

2.2.2 Le réveil dans les réseaux spéciaux

Les arbres. Le coût de l'utilisation du protocole *WFlood* pour le réveil dépendra du nombre d'initiateurs. En fait, s'il y a un seul initiateur, alors c'est juste une diffusion qui vaut seulement $n - 1$ messages. A l'opposé, si chaque entité démarre de manière indépendante, il y aura un total de $2(n - 1)$ messages. Soit k_* le nombre des initiateurs, qui n'est pas un paramètre système comme n ou m, mais limité cependant, un paramètre système : $k_* \leq n$. Le nombre total de messages lorsqu'on exécute *WFlood* dans un arbre sera exactement :

$$M[WFlood/Tree] = n + k_* - 2. \quad (2.5)$$

Les hypercubes étiquetés. Dans la section 2.1, nous avons pu, en exploitant certaines propriétés des hypercubes et l'étiquetage de dimension, construire un protocole qui utilise seulement $O(n)$ messages, au lieu de $\Omega(n \log n)$ messages requis par le protocole générique. Peut-on obtenir le même résultat pour le réveil ? En d'autres termes, peut-on exploiter les propriétés des hypercubes étiquetés pour améliorer le protocole générique ? La réponse est malheureusement Non.

Lemme 2.2.1 $M(Wake\text{-}Up/R; H) = \Omega(n \log n)$.

Comme conséquence, on pourrait utiliser le protocole *WFlood*, qui utilise $O(n \log n)$ messages. En résumé,

Propriété 2.2.3 *La complexité en messages du réveil sous* ***R*** *dans un hypercube à k dimensions avec étiquetage de dimension est* $\Theta(n \log n)$.

Graphes complets. L'utilisation du protocole générique *WFlood* requiert $O(n^2)$ messages. On peut, bien évidemment, utiliser le protocole simplifié *KBcast* développé pour les graphes complets. Le nombre de messages transmis sera $k_*(n - 1)$, où k_* représente le nombre des initiateurs. Même dans le cas le plus défavorable (lorsque chaque entité est réveillée de manière indépendante et qu'elles lancent simultanément le protocole, $O(n^2)$ messages seront transmis.

En exploitant les propriétés des graphes complets, peut-on construire un protocole de réveil qui utilise seulement $O(n)$ messages, au lieu de $O(n^2)$? Nous avons déjà réalisé cela pour la diffusion. Etonnamment, dans ce cas aussi, la réponse est Non.

Lemme 2.2.2 $M(Wake\text{-}Up/R\ ;\ K) = \Omega(n^2)$.

Ceci implique que *WFlood* est une solution optimale en message pour le réveil. En d'autres termes :

Propriété 2.2.4 *La complexité en message du réveil sous* **R** *dans un réseau complet est* $\Theta(n^2)$.

Graphes complets avec identificateurs distincts. Pour réduire le nombre de messages, un environnement avec plus de restrictions est requis; càd, nous avons besoin de faire d'autres suppositions.

Par exemple, si nous ajoutons la restriction que les entités ont des noms uniques (restriction Initial Distinct values (ID)), alors il existe des protocoles capables de réaliser le réveil en $O(n \log n)$ messages dans un graphe complet. Ils ne sont pas simple mais résolvent, en fait, un problème plus complexe qui est celui de l'élection (décrit plus loin). Curieusement, aucune autre amélioration n'est possible. En fait, si $\boldsymbol{IR + K = R \cup K}$; alors la complexité en message du réveil dans un graphe complet sous les restrictions standards R plus ID est comme suit :

Propriété 2.2.5 $M(\text{Wake-Up}/R; ID; K) \geq 0.5n \log n.$

Pour montrer pourquoi cela est vrai, nous construisons un cas mauvais mais possible, que tout protocole peut rencontrer, et nous montrons que dans un tel cas, $O(n \log n)$ messages seront échangés. La limite inférieure sera maintenue même en présence d'un ordonnancement des messages. Pour plus de simplicité, nous supposons que n est une puissance de 2 ; le résultat sera valable même si ce n'est pas le cas.

Pour construire la mauvaise solution pour un protocole arbitraire A, nous considérerons un *jeu* entre les entités d'un coté et un adversaire de l'autre : les entités obéissent aux règles du protocole et l'adversaire essayera de faire en sorte que le scénario le plus mauvais ait lieu de tel sorte à forcer les entités à utiliser autant de messages que possible. L'adversaire a les aptitudes suivantes :

1. Il décide des valeurs initiales des entités (qui sont cependant distinctes);

2. Il décide quelles entités lancent l'exécution de A spontanément, et quand;

3. Il décide quand un message transmis arrive (obligatoirement au bout d'un temps fini); et

4. Plus important, il décide de la correspondance entre les liaisons et les étiquettes : soient $e_1, e_2, \ldots, e_k$ les liaisons incidentes de x, et soient $l_1, l_2, \ldots, l_k$ les étiquettes des ports utilisées par x pour ces liaisons ; durant l'exécution, lorsque x réalise une commande "**send to** l", et l n'a pas été encore assigné, l'adversaire choisira à laquelle des liaisons non utilisée (i.e., à travers laquelle aucun message n'a été reçu ni envoyé) l sera assigné.

Remarque. L'envoi d'un message à plus d'un port sera traité comme l'envoi d'un message à chacun des ces ports, un à la fois (dans un ordre arbitraire).

Quelque soit la décision de l'adversaire, elle peut avoir lieu dans une exécution réelle. Considérons maintenant, à quel point un adversaire peut créer un cas mauvais pour A.

Deux ensembles d'entités seront dits *connectés* à l'instant *t* si au moins un message a été transmis d'une entité d'un ensemble vers une entité de l'autre ensemble.

Stratégie de l'adversaire.

1. Initialement, l'adversaire réveillera seulement une entité *s*, que nous appellerons *seed* (graine), qui lance l'exécution du protocole. Lorsque *s* décide d'envoyer un message sur le port de numéro *l*, l'adversaire réveillera une autre entité *y* et assigne le label *l* à l'arrête de *s* à *y*. Il décide, après, de retarder la transmission du message sur cette liaison jusqu'à ce que *y* décide aussi d'envoyer un message sur un certain port de numéro *l'*; l'adversaire assignera alors le label *l'* à la liaison partant de *y* à *s* et fait en sorte que les deux messages arrivent à leurs destinations simultanément. Dans ce cas chaque message atteindra un nœud réveillé et les deux entités sont connectées.

A partir de maintenant, l'adversaire agira de manière similaire en faisant en sorte que les messages arrivent toujours à des nœuds déjà réveillés ; et que l'ensemble des nœuds réveillés est connecté.

2. Considérons une entité *x* exécutant une opération **send** à un label *a* non assigné.

(a) Si *x* a une liaison non utilisée (i.e., à travers laquelle aucun message n'a été transmis) qui la connecte à un nœud non réveille, l'adversaire assignera *a* à cette liaison. En d'autres termes, l'adversaire essayera de faire en sorte qu'une entité réveillée envoi des messages à d'autres entités réveillées.

(b) Si toutes les liaisons entre *x* et les nœuds réveillés ont été utilisées, alors l'adversaire créera un autre ensemble de nœuds réveillés et connectera les deux ensembles.

i. Soit $x_0, \ldots, x_{k-1}$ les nœuds couramment réveillés, ordonnés selon le temps de réveil (ainsi, $x_0 = s$ est la graine, et $x_1 = y$). L'adversaire réalise la fonction suivante : choisit *k* nœuds non actifs $z_0, \ldots, z_{k-1}$; établir une correspondance entre x_j et z_j ; assigner les valeurs initiales aux nouvelles entités de tel sorte que l'ordre entre elles soit le même que celui entre les valeurs des entités correspondantes; réveille ces entités et les force à avoir la même exécution (même ordonnancement et même retards) comme l'ont fait celles correspondantes (Ainsi, z_0 sera réveillé le premier, son premier message sera envoyé à z_1, qui sera réveillé après et enverra un message à z_0, et ainsi de suite)

ii. L'adversaire assignera alors le label *a* à la liaison connectant *x* à son entité correspondante *z* dans le nouvel ensemble; le message sera maintenu en transit jusqu'à ce que *z* (comme la fait *x*) aura besoin de transmettre un message sur une liaison non utilisée (soit *b* son label) mais toutes les arrêtes le connectant à son ensemble d'entités réveillées ont toutes été déjà utilisées.

iii. Lorsque cela arrive, l'adversaire assignera le label *b* à la liaison de *z* à *x* et fait que les deux messages entre *x* et *z* arrivent et sont traités.

Résumons la stratégie de l'adversaire : Il essaye de forcer le protocole à envoyer les messages aux entités déjà réveillées et réveille de nouvelles entités uniquement lorsqu'il ne peut pas faire autrement. Les entités réveillés récemment sont au même nombre que les entités déjà réveillées et sont forcées par l'adversaire à avoir la même exécution entre elles comme l'ont fait les autres entités avant qu'une quelconque communication n'ait lieu entre les deux ensembles. Lorsque cela arrive, on dira que l'adversaire a lancé une nouvelle phase.

Examinons maintenant les situations créées par l'adversaire et analysons le coût du protocole lors des exécutions correspondantes. Soit *Active(i)* les entités réveillées en phase *i* et *New(i)=Active(i)− Active(i−1)* les entités que l'adversaire a réveillé dans cette phase. Initialement, *Active*(0) est juste la graine. Les entités récemment réveillées sont égales en nombre à celles déjà réveillées ; càd, *|New(i)| = |Active(i − 1)|.*

Soit *μ(i − 1)* le nombre *total* de messages échangés avant l'activation des nouvelles entités. L'adversaire force les nouvelles entités à avoir le même exécution comme l'ont fait les entités de *Active(i − 1)*, donc, échangeant *μ(i − 1)* messages, avant de permettre aux deux ensembles de se connecter. Ainsi, le nombre total de messages avant la communication entre les deux ensembles est *2μ(i − 1)*.

Une fois la communication a lieu, combien de messages (incluant ces deux courants) sont–ils transmis avant la nouvelle phase ? La réponse exacte dépend du protocole *A*, cependant, indépendamment du protocole utilisé, l'adversaire ne lancera une nouvelle phase *i + 1* que lorsqu'il est forcé à cela; càd, quand *x* exécute une commande *"send to l"* (où *l* est un label non assigné) et toutes les liaisons connectant *x* aux autres entités réveillées ont déjà été utilisées. Ceci signifie que *x* a du soit envoyer à/ recevoir de toutes les autres entités dans *Active(i) = Active(i−1) ∪ New(i).* Supposons que $x \in$ *Active(i − 1)*; alors, parmi tous ces messages, ceux entre *x* et *New*(*i*) ont eu lieu durant la phase *i* (vu que ces entités n'étaient pas actives avant); ce qui signifie qu'au moins *|New(i)| = |Active(i −1)|* messages additionnels seront transmis avant la phase *i + 1*. Si par contre, $x \in$ *New*(*i*), ces messages ont tous été transmis dans cette phase (*x* n'été pas réveillé avant). En d'autres termes, même dans ce cas, *|New(i)| = |Active(i −1)|* messages additionnels seront transmis avant la phase *i + 1*.

En résumé, le coût total *μ(i − 1)* avant la phase *i* est ainsi doublé et au moins *|Active(i −1)|* messages additionnels seront transmis avant la phase *i + 1*. En d'autres termes, $\mu(i) \geq 2\,\mu(i-1) + |Active(i-1)|$.

Comme les entités réveillées doublent à chaque phase, et initialement seule la graine est active, alors *|Active(i)| = 2i* . Donc, partant de $\mu(0) = 0$,

$$\mu(i) \geq 2\,\mu(i-1) + 2^{i-1} \geq i\,2^{i-1}$$

Le nombre total de phases sera exactement log n comme les processus réveillés double chaque phase.

Ainsi, avec cette stratégie, l'adversaire peut forcer n'importe quel protocole à transmettre *au moins* $\mu(\log n)$ messages. Vu que $\mu(\log n) \geq 0.5\, n \log n$, il s'en suit que n'importe quel protocole de réveil transmettra $\Omega(n \log n)$ messages dans le pire des cas même si les entités ont des identificateurs distincts.

Des protocoles plus efficaces peuvent être construits si nous avons dans notre système un bon étiquetage des liaisons.

2.3 PARCOURS DE RESEAUX

Le parcours de réseau permet à toutes les entités du réseau d'être visitées séquentiellement (l'une après l'autre). Ses utilisations principales sont le contrôle et la gestion de ressource partagée et la recherche séquentielle. En termes plus abstraits, le problème de parcours débute avec une configuration initiale où toutes les entités sont dans le même état (appelons le *unvisited*) excepté celui qui est visité et qui est le seul initiateur ; le but étant de rendre toute les entités visitées mais séquentiellement (i.e., une à la fois).

Un protocole de parcours (*traversal protocol*) est un algorithme distribué qui, partant d'un seul initiateur, permet à un message spécial appelé jeton de parcours (*traversal token*) ou simplement jeton, d'atteindre chaque entité séquentiellement (i.e., one i.e., une à la fois). Une fois un nœud atteint par le jeton, il est marqué comme *visité*. En fonction de la stratégie de parcours employée, on aura différents protocoles de parcours.

2.3.1 Parcours en profondeur d'abord

Une stratégie de parcours de graphes bien connue est le parcours en profondeur d'abord (*depth-first traversal*). Selon cette stratégie, le graphe est visité (i.e., le jeton et transmis) en essayant d'aller droit le plus possible, s'il est transmis à un nœud déjà visité, il est renvoyer à son émetteur et la liaison en question est marquée *back-edge*; si le jeton ne pas plus être transmis (il s'agit d'un nœud où tous ses voisins ont été visités), l'algorithme effectue un retour arrière ("backtrack") jusqu'à ce qu'il trouve un nœud non visité auquel le jeton sera transmis. L'implémentation distribuée de la stratégie en profondeur d'abord est directe.

1. Lorsqu'elle est visitée pour la première fois, l'entité mémorise qui a envoyé le jeton, crée une liste de ses voisines non encore visitées, transmet le jeton à l'une d'elle (l'enlève de la liste), et attend la réponse qui retourne le jeton.

2. Lorsqu'un nœud voisin reçoit le jeton, il retourne le jeton immédiatement s'il a été déjà visité par quelqu'un d'autre, notifiant que la liaison est un backedge; sinon, il transmet d'abord le jeton à chacun de ses voisins non visités séquentiellement puis répond en retournant le jeton.

3. À la réception d'une réponse, l'entité transmet le jeton à un autre voisin non visité.

4. Lorsqu'il n'y aura plus de voisin non visités, l'entité ne pourra plus transmettre le jeton, elle transmet alors une réponse retournant le jeton à l'entité émettrice d'où elle a reçu initialement le jeton.

Remarque. Lorsque un voisin dans l'étape (2) détermine qu'une liaison est un back-edge, il sait que l'émetteur du jeton est déjà visité ; ainsi, il l'enlève de la liste des voisins non visités.

On utilisera trois types de message : "T" pour transmettre le jeton lors du parcours, "Backedge" pour notifier une détection d'un back-edge, et "Return" pour retourner le jeton après la terminaison locale.

Le protocole *DFTraversal* est montré dans la figure 2.6, où l'opération d'extraction d'un élément de l'ensemble B et son assignation à une variable a est notée $a \Leftarrow B$. Examinons ses coût.

Considérons une liaison $(x,y) \in E$. Quel message peut-on envoyer sur celle-ci? Supposons que x envoi T à y; alors y n'enverra à x soit un Return (s'il était libre lorsque T est arrivé) ou *Backedge*, sinon. En d'autres termes, sur chaque liaison, il y aura exactement deux messages transmis. Vu que le parcours est séquentiel, **T**[*DFTraversal*]=**M**[*DFTraversal*]; donc :

$$T[DFTraversal] = M[DFTraversal] = 2m. \quad (2.6)$$

Pour déterminer l'efficacité du protocole, nous allons déterminer la complexité du problème. En utilisant exactement la même technique que celle utilisée dans la preuve du théorème 2.1.1, nous avons :

Théorème 2.3.1 *$M(DFT/R) \geq m$.*

Donc, le coût $2m$ message du protocole *DFTraversal* est en fait excellent et le protocole est optimal en messages.

Propriété 2.3.1 *La complexité en message du parcours en profondeur d'abord sous R est $\Theta(m)$.*

Les exigences sur le temps du parcours en profondeur d'abord sont assez différentes de ceux de la diffusion. En fait, vu que chaque nœud doit être visité séquentiellement, en partant de l'unique initiateur, la complexité en temps est au moins égale au nombre de nœuds.

```
PROTOCOL DFTRAVERSAL.
        • STATUS: S = {INITIATOR,IDLE,VISITED,DONE};
          SINIT = {INITIATOR,IDLE}; STERM = {DONE}.
        • RESTRICTIONS: R ;UI.
  INITIATOR
        SPONTANEOUSLY
            BEGIN
              UNVISITED:= N(X);
              INITIATOR:= TRUE;
              VISIT;
            END
  IDLE
        RECEIVING (T )
            BEGIN
              ENTRY: = SENDER;
              UNVISITED: = N(X) − {SENDER};
              INITIATOR: = FALSE;
              VISIT;
            END
  VISITED
        RECEIVING (T )
            BEGIN
              UNVISITED: = UNVISITED −{SENDER};
              SEND(BACKEDGE) TO {SENDER};
            END
        RECEIVING(RETURN)
            BEGIN
              VISIT;
            END
        RECEIVING(BACKEDGE)
            BEGIN
              VISIT;
            END
  PROCEDURE VISIT
  BEGIN
     IF UNVISITED ≠ ∅ THEN
            NEXT ⇐ UNVISITED;
            SEND(T) TO NEXT;
            BECOME VISITED
     ELSE
            IF NOT(INITIATOR) THEN SEND(RETURN) TO ENTRY; ENDIF
            BECOME DONE;
     ENDIF
  END
```

FIGURE 2.6: Protocole DFTraversal

Théorème 2.3.2 *T (DFT/R) ≥ n − 1.*

La complexité du protocole *DFTraversal* est très intéressante. En fait, la limite supérieure $2m$ peut être plus grande de plusieurs ordres de grandeur que la limite inférieure $n - 1$. Par exemple, dans un graphe complet, $2m = n^2 - n$. Des améliorations significatives dans la complexité en temps peuvent, cependant, être fait en allant vers une granularité plus fine. Nous discutons cela dans ce qui suit.

2.3.2 Le Hacking

Examinons le protocole *DFTraversal* pour voir si il peut être amélioré en temps.

IMPORTANT. Lorsqu'on mesure le temps idéal, nous considérons seulement les exécutions synchrones ; cependant, lorsqu'on mesure le nombre de message et on cherche à vérifier si un protocole est correct, nous considérons tout ordonnancement possible des événements et en particulier l'exécution non synchrone.

Le hacking de base. Le protocole que nous avons construit est totalement séquentiel : dans une exécution synchrone, à chaque unité de temps seul un message sera envoyé et chaque message requière une unité de temps. Donc, pour améliorer la complexité en temps, nous avons besoin (1) de réduire le nombre de messages et/ou (2) introduire une certaine concurrence. Par définition du parcours, chaque entité doit recevoir le jeton (message T) au moins une fois. Dans l'exécution de notre protocole, certaines entités peuvent le recevoir deux fois. Les liaisons sur lesquelles arrive le jeton sont précisément les backedges.

Question. Peut-on éviter l'envoi des messages T sur les backedges?

Pour répondre à cette question, nous devons comprendre pourquoi les messages T sont envoyés sur les backedges. Lorsqu'une entité x transmet un message T à y, elle ne sait pas si la liaison est un backedge ou non, càd, si y a déjà été visité par quelqu'un d'autre ou non. Si x savait qui de ses voisins ont déjà été visités, il n'aurait pas envoyé des messages T à ces derniers, on n'aurait plus besoin des messages Backedge retournés, et nous aurions pu éviter une perte de temps et l'envoi de messages inutiles.

Supposons que, à chaque fois qu'un nœud est visité (i.e., reçoit T) pour la première fois, il notifie ses (autres) voisins de l'occurrence de cet événement (e.g., envoyant un message "Visited") et attend un acquittement (e.g., réception d'un message "Ack" message) d'eux avant de leur transmettre le jeton. La conséquence d'un acte aussi simple est que, maintenant, un noeud prêt à transmettre le jeton sait réellement qui de ses voisins a déjà été visité. C'est exactement ce qu'on souhaite. Le prix à payer est la transmission des messages Visited et Ack.

Notons maintenant que une entité libre (*idle* : càd non encore impliquée dans le parcours) peut recevoir un message Visited comme premier message. Dans le nouveau protocole, nous ferons en sorte qu'une telle entité entre dans un nouveau statut que nous appelons *available*. Examinons les effets de ce changement sur coût en temps du protocole. Appelons *DF+* le nouveau protocole. Le temps est déterminé par les messages séquentiels. Il y a quatre type de messages: T, Return, Visited, et Ack.

Chaque entité (excepté l'initiatrice) recevra seulement un message T et envoi seulement un message Return; l'initiatrice ne reçoit aucun message T ni envoi de message Return; ainsi, au total il y aura $2(n-1)$ messages. Vu que toutes ces communications ont lieu séquentiellement (i.e., sans chevauchement), le temps nécessaire pour envoyer les messages T et recevoir les messages Return sera $2(n-1)$.

Pour déterminer combien d'unités de temps idéal sont nécessaire pour la transmission de messages Visited et Ack, considérons une entité : la transmission de tous les messages Visited prend seulement une unité de temps, vu qu'ils sont envoyés simultanément. Les messages Ack correspondants seront envoyés aussi simultanément, ajoutant une autre unité de temps. Vu que chaque nœud fera la même chose, l'envoi des messages T et la réception des Ack augmenteront le temps idéal de l'algorithme originel par exactement $2n$. Ceci nous donne un coût en temps de

$$T[DF+] = 4n - 2. \quad (2.7)$$

Il est aussi simple de calculer combien de messages coûte ce nouveau protocole. Comme mentionner avant il y a un total de $2(n-1)$ message T et Return. En plus chaque entité (sauf l'initiatrice) envoie un message Visited à toutes ses voisines sauf celle d'où elle a reçu le jeton, l'initiatrice enverra ce message à toutes ses voisines. Ainsi, en notant par s l'initiatrice, le nombre total de messages Visited est $|N(s)| + \Sigma_{x \neq s} (|N(x)| - 1) = 2m - (n-1)$. Vu que pour chaque message Visited il y aura un Ack, le coût total en messages sera

$$M[DF+] = 4m - 2(n-1) + 2(n-1) = 4m. \quad (2.8)$$

En résumé, nous avons été capable de réduire le coût en temps de $O(m)$ à $O(n)$. Le prix de cette réduction est la multiplication par 2 du nombre de messages (ceci est dû au fait que le théorème 2.3.2, est optimal).

Propriété 2.3.2 *La complexité en temps idéal du parcours en profondeur d'abord sous R est* $\Theta(n)$.

Le hacking avancé. Voyons si le nombre de messages peut être réduit sans augmenter significativement le coût en temps.

Question. Peut-on éviter l'envoi des messages Ack?

Supposons qu'on n'envoie pas de Ack et que l'entité *x* a envoyé le message Visited à ses voisines, *x* continuera immédiatement en envoyant le jeton. Supposons que après un certains temps, le jeton arrive pour la première fois à l'entité voisine *z* (voir Fig. 2.7); il est possible que le message Visited envoyé par x à z ne soit pas encore arrivé (suite à des retards dans la communication). Dans ce cas, *z* ne sait pas que *x* est déjà visité et lui envoie le message T. Càd, nous envoyons encore un message T sur backedge ce que nous avons évité par le protocole amélioré précedent.

Cependant, l'algorithme est différent (on n'envoie que des messages Visited) et la situation décrite n'arrivera pas toujours. Même si cette situation arrive, *z* se rendera compte de son erreur et continuera comme s'il n'avait pas envoyé le message T. De même pour *x*.

Même la validité de l'algorithme n'est pas mise en cause, les erreurs apportent un coût additionnel en messages. Appelons ce protocole modifié DF++ et examinons son coût. Comme précédemment les T et Return corrects coûtent $2n - 2$ messages, et les messages Visited sont au nombre de $2m - n + 1$ au total. Il faut maintenant adjoindre les messages d'erreur ; chaque erreur coûte un message. Le nombre d'erreurs peut être très grand. En fait, les retards peuvent forcer des erreurs sur chaque backedge; et sur certains il peut y avoir deux erreurs, une dans chaque direction.

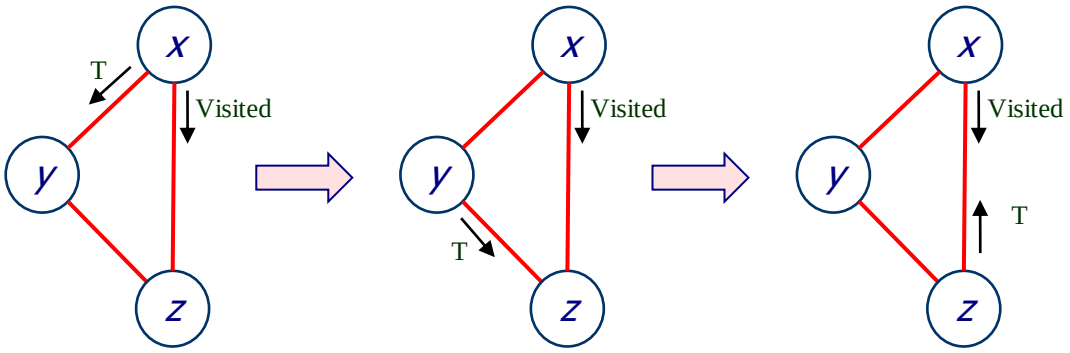

FIGURE 2.7: Erreur de transmission du message T.

En d'autres termes il y aura au plus $2(m - n + 1)$ messages T incorrects. Au total, nous aurons :

$$\boldsymbol{M}[DF{+}{+}] \leq 4m - n + 1$$

Concernant le temps, nous avons réalisé une amélioration par l'élimination des Ack, ce qui nous permet de réduire le temps de *n* unités. Puisque l'entité n'a plus à attendre des Ack, elle peut transmettre le jeton T simultanément avec l'envoi des messages Visited. Si l'entité n'a pas de voisines, elle enverra le message Return en même temps que visited. Ainsi, l'envoi de Visited se fera toujours en même temps avec l'envoi de T ou du Return, ce qui réduit le temps de *n* autres unités.

Sans considérer les erreurs, le temps total sera $2n - 2$. Considérons maintenant, le coût en temps des erreurs. On constate, en considérant le temps idéal, qu'aucune erreur n'a lieu, car les erreurs sont causées par les délais de communication arbitrairement longs ce qui n'est pas le cas avec le temps idéal.

Ainsi, $\boldsymbol{T}[DF{+}{+}] = 2n - 2$

Le Hacking Extrème. Il s'agit d'utiliser le message T comme un message Visited implicite.

$$\boldsymbol{T}[DF_*] = 2n - 2$$

$$\boldsymbol{M}[DF_*] \leq 4m - 2n + f_* + 1$$

f_* est un paramètre non système qui correspond au nœuds feuilles qui n'ont pas à envoyer le message Visited.

PROTOCOL DF*

- **STATUS: S = {INITIATOR,IDLE,AVAILABLE,VISITED,DONE};**
  SINIT = {INITIATOR,IDLE}; STERM = {DONE}.
- **RESTRICTIONS: R ;UI.**

INITIATOR
  SPONTANEOUSLY
    BEGIN
      INITIATOR:= TRUE;
      UNVISITED:= N(X);
      NEXT ⇐ UNVISITED;
      SEND(T) TO NEXT;
      SEND(VISITED) TO N(X)-{NEXT};
      BECOME VISITED
    END

IDLE
  RECEIVING(T)
    BEGIN
      UNVISITED:= N(X);
      FIRST-VISIT;
    END
  RECEIVING(VISITED)
    BEGIN
      UNVISITED:= N(X) − {SENDER};
      BECOME AVAILABLE
    END

AVAILABLE
    RECEIVING(T)
      FIRST-VISIT;
    RECEIVING(VISITED)
      BEGIN
        UNVISITED:= UNVISITED − {SENDER};
      END

VISITED

```
        RECEIVING(VISITED)
            BEGIN
                UNVISITED:= UNVISITED −{SENDER};
                IF NEXT = SENDER THEN VISIT; ENDIF
            END
        RECEIVING(T)
            BEGIN
                UNVISITED:= UNVISITED −{SENDER};
                IF NEXT = SENDER THEN VISIT; ENDIF
            END
        RECEIVING(RETURN)
            BEGIN
                VISIT;
            END
```

FIGURE 2.8: Protocole DF*

```
PROCEDURE FIRST-VISIT
BEGIN
    INITIATOR:= FALSE;
    ENTRY:=SENDER;
    UNVISITED:= UNVISITED-{SENDER};
    IF UNVISITED ≠ ∅ THEN
        NEXT ⇐ UNVISITED;
        SEND(T) TO NEXT;
        SEND(VISITED) TO N(X)−{ENTRY,NEXT};
        BECOME VISITED;
    ELSE
        SEND(RETURN) TO {ENTRY};
        SEND(VISITED) TO N(X)−{ENTRY};
        BECOME DONE;
    ENDIF
END
PROCEDURE VISIT
BEGIN
    IF UNVISITED ≠ ∅ THEN
        NEXT ⇐ UNVISITED;
        SEND(T) TO NEXT;
    ELSE
        IF NOT(INITIATOR) THEN SEND(RETURN) TO ENTRY; ENDIF
        BECOME DONE;
    ENDIF
END
```

FIGURE 2.9: Routines utilisées par le Protocole DF*

2.3.3 Parcours dans les réseaux spéciaux

Les arbres. Le parcours DF est optimale en temps et en messages car il n'y a pas de cycle ce qui élimine les messages Backedge. Notons que le parcours en profondeur d'abord construit un anneau virtuel ou certains nœuds apparaissent plus d'une fois.

***M**[DF Traversal/Tree] = **T**[DF Traversal/Tree] = 2n − 2*

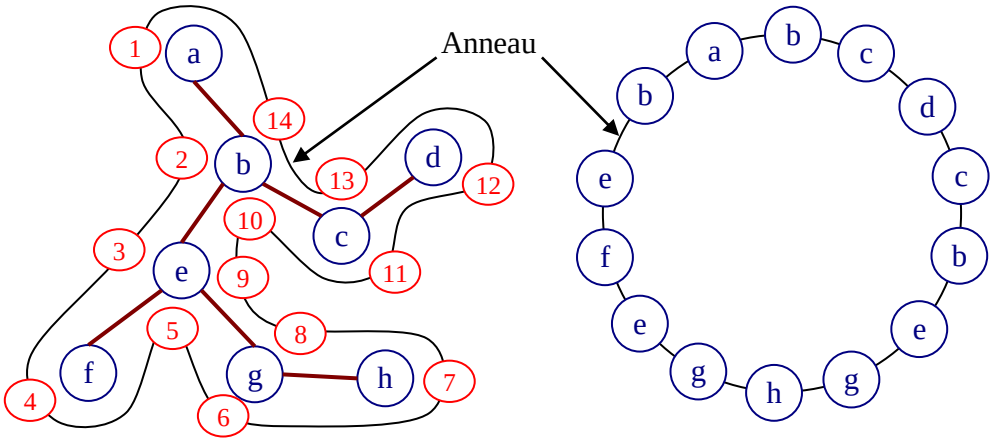

FIGURE 2.10: Anneau virtuel créé par le parcours DF.

Les anneaux. Dans un anneau, chaque nœud a deux voisins exactement. Le parcours en profondeur d'abord peut être changé, l'initiateur choisit une direction et le jeton est transmis selon cette direction d'un nœud à l'autre. Le message Ret n'est plus utilisé et le parcours se termine lorsque le jeton revient au nœud initiateur. Le coût en temps et en messages vaut exactement n.

Les graphes complets. L'execution de DF* nécessite $O(n^2)$ messages car $m=n^2-n$. En exploitant la connaissance que le réseau est un graphe complet il est possible d'améliorer DF* : l'initiateur transmet séquentiellement le jeton à tous ses voisins et chcun des voisins se contentera de retourner le jeton. Le nombre total de messages vaut $2(n-1)$, et de même pour le temps.

2.3.4 Considérations sur le parcours de réseaux

On utilise le parcours principalement dans le contrôle et la gestion des ressources partagées. Le jeton devient alors un synonyme de persmission d'utiliser la ressource. Cependant notons que l'accès aux ressources et continu alors que le parcours se termine.

On constate aussi que le parcours résout le problème de diffusion mais l'inverse n'est pas vrai. La diffusion par parcours est plus coûteuse en temps et en messages que la stratégie d'inondation.

2.4 Construction d'une Arbrescence Couvrante (Spanning Tree)

Dans la pratique beaucoup d'investissements sont consentis dans la construction des liaisons et on constate que le nombre de liaisons est largement supérieur au nombre de nœuds. D'un autre côté, la complexité des algorithmes de base tels que Diffusion,

Réveil et Parcours dépend du nombre de liaisons. En d'autres termes, plus il y a de liaisons, plus le coût est important. La complexité peut atteindre $O(n^2)$.

Il se pose alors la question : Comment peut-on réduire le coût sans poser des restrictions suplémentaires sur les protocoles ? (càd sans réduire leur applicabilité).

La solution consiste à :

1. Construire un sous réseau G' de G et
2. Exécuter les protocoles uniquement sur G'

Notons que si G' est connecté et couvre tout G (càd contient tous les nœuds de G) alors résoudre un problème sur G' le résout effectivement sur G. Par exemple la diffusion sur G' est équivalente à la diffusion sur G, le parcours sur G' est équivalent au parcours sur G, et ainsi de suite.

Si on souhaite réduire le coût au maximum, il faut construire un sous-réseau G' avec un minimum de liaisons tout en préservant la couverture de G (tous les nœuds sont compris) est la connexion. Ainsi, notre choix se porte sur l'arbre car m=n-1 est la plus petite valeur possible pour le nombre de liaisons.

L'approche consiste donc à

1. Construire un arbre G' courant G et à
2. Exécuter les protocoles sur G'

En absence de pannes, la construction de G' se fait en une seule fois et le côut final doit inclure le coût de la construction de l'arborescence couvrante et le coût des protocoles qui seront exécutés sur celle-ci.

2.4.1 Construction d'une Arborescence couvrante avec un initiateur unique

La construction d'une arborescence couvrante, nous permet de partir d'une configuration initiale où :

1. chaque entité connaît ses voisines à une configuration où chaque entité x a un sous-ensemble de voisines notées $Tree\_neighbors(x) \subseteq N(x)$
2. La collection de liaisons correspondantes forme une arborescence couvrante de G.

La construction de l'arborescence couvrante revient à chercher un algorithme distribué qui spécifie ce que chaque nœud doit faire en recevant un message dans un état

donné, de telle sorte que une fois exécuté nous obtenons une arborescence couvrante T(G) de G que nous notons simplement par T.

Remarque. T est inconnu aux entités au départ et après sa construction et nous considererons les restrictions standards dans la construction de T.

Stratégie de construction : Demander aux voisins (*Ask-Your-Neighbors*)

1. L'initiatrice *s* demande à toutes ses voisines en envoyant un message *Q* qui signifie (*"Etes vous mon voisin dans l'arborescence couvrante"?*).

2. Une entité $x \neq s$ répondera *"Oui"* uniquement lorsqu'elle reçoit le message la première fois et à cette occasion, elle posera la question, à son tour, à toutes ses voisines. Si l'entité x reçoit le message pour la seconde fois (voire troisième fois et plus), elle répondera par *"Non"*. L'initiatrice répondera toujours par *"Non"*.

3. Chaque entité termine lorsqu'elle aura réçu une réponse de toutes ses voisines à qui elle a posé la question.

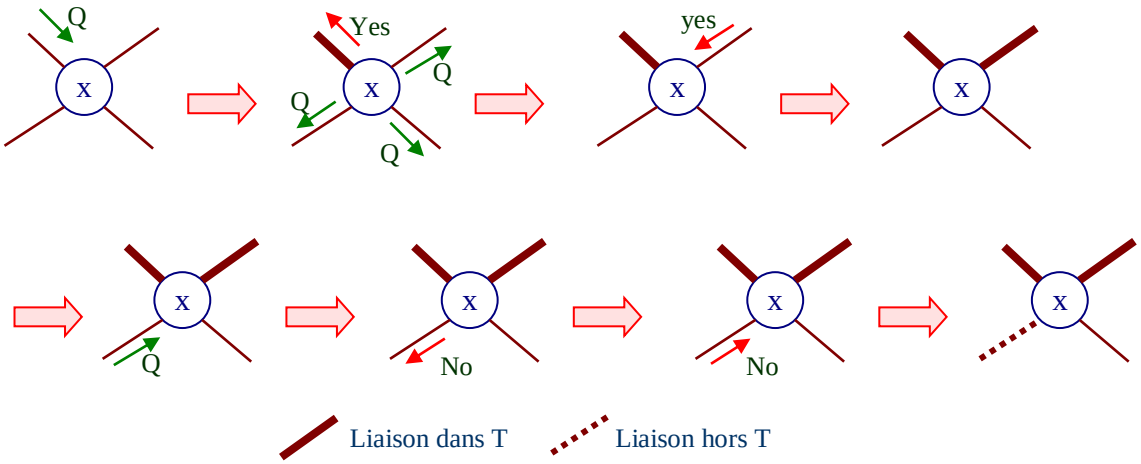

FIGURE 2.11: Mise en œuvre de la stratégie.

Il aisé de constater que cette stratégie n'est autre qu'une innondation ou la réception de chaque message d'information est faite par un accusé de réception (Flood + Reply).

Le protocole de la figure 2.12 que nous appelons Shout montre une implémentation de la stratégie de construction d'une arborescence couvrante.

PROTOCOL SHOUT
- STATUS: S = {INITIATOR,IDLE,ACTIVE,DONE};
  SINIT = {INITIATOR,IDLE};
  STERM = {DONE}.
- RESTRICTIONS: R ;UI.

INITIATOR
  SPONTANEOUSLY

```
            BEGIN
                ROOT:= TRUE;
                TREE-NEIGHBORS:=∅;
                SEND(Q) TO N(x);
                COUNTER:=0;
                BECOME ACTIVE;
            END
IDLE
        RECEIVING(Q)
        BEGIN
            ROOT:= FALSE;
            PARENT:= SENDER;
            TREE-NEIGHBORS:={SENDER};
            SEND(YES) TO {SENDER};
            COUNTER:=1;
            IF COUNTER=|N(x)| THEN
                BECOME DONE
                ELSE
                SEND(Q) TO N(x) – {SENDER};
                BECOME ACTIVE;
            ENDIF
        END
ACTIVE
        RECEIVING(Q)
            BEGIN
                SEND(NO) TO {SENDER};
            END
        RECEIVING(YES)
            BEGIN
                TREE-NEIGHBORS:=TREE-NEIGHBORS ∪{SENDER};
                COUNTER:=COUNTER+1;
                IF COUNTER=|N(x)| THEN BECOME DONE; ENDIF
            END
        RECEIVING(NO)
            BEGIN
                COUNTER:=COUNTER+1;
                IF COUNTER=|N(x)| THEN BECOME DONE; ENDIF
            END
```

FIGURE 2.12: Le protocole Shout

Remarque

Il est important de noter que dans le protocole Shout la terminaison est locale est aucune entité n'est au courant de la terminaison globale. Cela est souvent le cas dans les algorithmes distribués. Cependant, l'entité initiatrice a besoin de savoir si la construction est finie pour lancer éventuellement un protocole de calcul.

Le staratégie proposée étant un *Flood+Reply* son coût en message vaut deux fois le coût du protocole d'inondation :

$$\boldsymbol{M}[Flood+Reply] = 2 \times \boldsymbol{M}[Flooding].$$

Le coût en temps vaut :

$$\boldsymbol{T}[Flood+Reply] = \boldsymbol{T}[Flooding]+1$$

Ce coût s'explique par le fait que les messages «yes» sont envoyés en même temps que les messages Q, sauf pour les nœuds feuilles qui ont un seul voisin

Pour le protocole Shout,

$$\mathbf{M}[Shout] = 4m - 2n + 2$$

$$\mathbf{T}[Shout] = r(s_*) + 1 \leq d + 1 \quad \text{où r désigne le rayon à partir de s}$$

Notons que la complexité du problème vaut :

$$M(\mathbf{SPT/RI}) \geq m.$$

La démonstration étant la même que celle de la diffusion. Quant à la complexité en temps idéal, en considérant n'importe quelle initiatrice, elle vaut :

$$T(\mathbf{SPT/RI}) \geq d.$$

2.4.2 Construction d'SPT avec initiateurs multiples

Théorème : Le problème de la construction de l'arborescence couvrante sous les restrictions R n'a pas de solution déterministe. Autrement dit il n'existe pas de protocole qui se termine toujours correctement au bout d'un temps fini.

Pour démontrer ce théorème, considérons l'arbre donné en figure 2.13 où les trois entités x, y et z sont des initiatrices et considérons une exécution synchrone (les protocoles exécutent exactement les mêmes actions au même moment) où les retards sont unitaires.

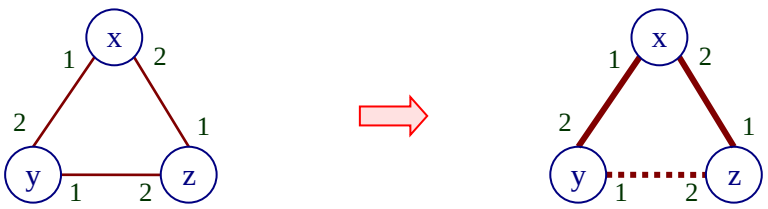

FIGURE 2.13: Exemple simple d'arbre.

Si on suppose l'existence d'un protocole A capable de construire une arborescence couvrante, alors à chaque étape de ce protocole, x, y et z passent dans un nouvel état qui est identique pour les trois entités et exécutent les mêmes actions qui conduisent à un nouvel état identique pour les trois entités et ainsi de suite.

On a besoin d'éliminer une seule liaison dans le réseau pour créer une arborescence couvrante dans le cas de la figure 2.13. Si on considère le protocole Shout, l'élimination de la liaison 1 par x signifie aussi l'élimination de la liason 1 par y et aussi par z ce qui crée une forêt couvrante déconnetée et non par une arborescence couvrante (figure 2.14).

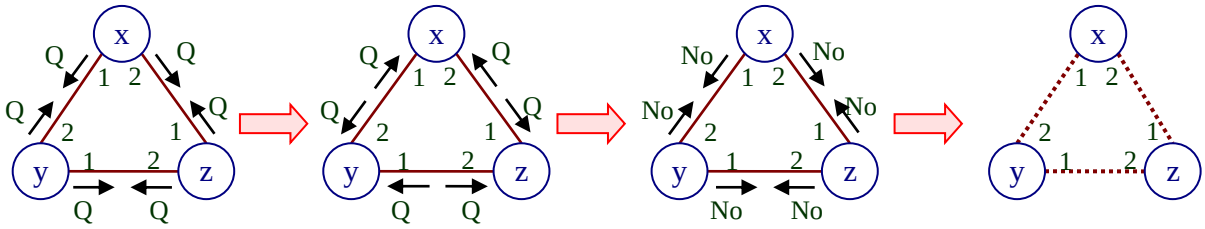

FIGURE 2.14: Création d'une forêt couvrante.

Pour trouver une solution, il faut ajouter d'autres restrictions : Une des restriction courante est d'associer des valeurs initiales distinctes à chaque nœud qu'on appelle aussi identificateurs ou noms globaux.

Avec cette nouvelle restriction, il est possible d'adapter le protocole Shout pour la construction d'une arborescence couvrante avec plusieurs initiatrices.

Shout tel qu'il est crée une forêt couvrante et non pas une arborescence. Pour remédier à cela, on laisse chaque initiatrice construire son arborescence en ajoutant avec chaque message l'identificateur de l'entité initiatrice. En d'autres termes, on crée autant d'arborescences couvrantes que d'initiatrice. Cela suppose une adaptation du protocole Shout pour avoir une multitude de variables (autant que d'initiatrices).

Vu que la construction de plusieurs arborescence est couteûse, on supprime lors de la construction toutes les arborescences sauf celle dont l'identificateur de l'initiatrice est le plus petit (voire le plus grand).

Pour savoir que la terminaison est atteinte, on utilise une diffusion sur l'arborescence couvrante.

Les figures 2.14 et 2.15 donnent le protocole qui permet de construire une arborescence couvrante en présence de plusieurs initiatrices.

```
PROTOCOL MULTISHOUT
    • STATUS: S = {IDLE, ACTIVE, DONE};
              SINIT = {IDLE}; STERM = {DONE}.
    • RESTRICTIONS: R ;ID.
IDLE
    SPONTANEOUSLY
        BEGIN
            ROOT:= TRUE;
            ROOT ID:=V(X);
            TREE NEIGHBORS:=∅;
            SEND(Q,ROOT ID) TO N(X);
            COUNTER:=0;
            CHECK COUNTER:=0;
            BECOME ACTIVE;
        END
    RECEIVING(Q,ID)
        BEGIN
            CONSTRUCT;
        END
ACTIVE
    RECEIVING(Q,ID)
        BEGIN
            IF ROOT ID = ID THEN
            COUNTER:=COUNTER+1;
            IF COUNTER=|N(X)| THEN DONE:= TRUE; CHECK; ENDIF
            ELSE
            IF ROOT ID > ID THEN CONSTRUCT;
            ENDIF
        END
    RECEIVING(YES, ID)
        BEGIN
            IF ROOT ID = ID THEN
            TREE-NEIGHBORS:=TREE-NEIGHBORS ∪ {SENDER};
            COUNTER:=COUNTER+1;
            IF COUNTER=|N(X)| THEN DONE:= TRUE; CHECK; ENDIF
            ENDIF
        END
    RECEIVING(CHECK, ID)
        BEGIN
            IF ROOT ID = ID THEN
              CHECK COUNTER:=CHECK COUNTER+1;
              IF (DONE ∧ CHECK_COUNTER=|CHILDREN|) THEN TERM;
              ENDIF
            ENDIF
        END
    RECEIVING(TERMINATE)
        BEGIN
            SEND(TERMINATE) TO CHILDREN;
            BECOME DONE;
        END
```

FIGURE 2.16: Protocol MultiShout

```
PROCEDURE CONSTRUCT
BEGIN
ROOT:= FALSE;
ROOT ID:= ID;
TREE NEIGHBORS:={SENDER};
PARENT:= SENDER;
SEND(YES,ROOT ID) TO {SENDER};
COUNTER:=1;
CHECK_COUNTER:=0;
IF COUNTER=|N(X)| THEN
        DONE:= TRUE;
        CHECK;
ELSE
        SEND(Q,ROOT-ID) TO N(X) − {SENDER};
ENDIF
BECOME ACTIVE;
END
PROCEDURE CHECK
BEGIN
CHILDREN:= TREE NEIGHBORS-{PARENT};
IF CHILDREN = ∅ THEN
        SEND(CHECK,ROOT ID) TO PARENT;
ENDIF
END
PROCEDURE TERM
BEGIN
IF ROOT THEN
SEND(TERMINATE) TO TREE-NEIGHBORS;
        BECOME DONE;
ELSE
        SEND(CHECK,ROOT-ID) TO PARENT;
ENDIF
END
```

FIGURE 2.15: Routines de MultiShout

2.5 CALCULS DANS LES RESEAUX EN ARBRE

Nous considérons les calculs sous les restrictions standard **R** avec une connaissance commune que le réseau est un arbre. Cette nouvelle restriction suppose que chaque nœud est capable de savoir s'il est une feuille de l'arbre (a un seul voisin) ou un nœud interne (a plus d'un voisin). Nous allons considérer ci après un protocole général qui permet d'implémenter divers protocoles de calcul sur les arbres. Il s'agit de la Saturation.

2.5.1 La saturation: Une technique de base

La saturation totale se compose de trois phases et est lancée par un nombre quelconque d'entité initiatrices.

1. La phase d'*activation*, lancée par les initiatrices, elle active toutes les entités;

2. La phase de *saturation*, lancée par les feuilles, elle conduit à la sélection d'un seul couple d'entités voisines; et

3. la phase de *resolution*, lancée par le couple d'entités sélectionné en 2.

La phase d'*activation* n'est autre qu'un réveil: chaque entité initiatrice envoie un message d'activation (i.e., réveil) à toutes ses voisines et passe dans l'état actif. Lors de la réception du message par une entité non initiatrice, elle le transmet à toutes ses voisines et devient active. Les entités déjà actives igniorent les messages d'activation. Au bout d'un temps fini toutes les entités seront réveillées.

Chaque entité feuille lance la phase de saturation en envoyant un message nommé M à son seul voisin que nous désignons par "parent," et passe dans l'état *processing*. (Notons que : Les messages M arrivent vers les nœuds internes au bout d'un temps fini). Un nœud interne attend la réception de messages M de tous ses voisins sauf un puis envoi, à son tour le message M à ce dernier voisin (nous le désignons par "parent") et passe dans l'état *processing*. Si un nœud M reçoit un message M de son Parent, il passe dans l'état *saturated*.

La phase de résolution est lancée par les nœuds saturés (seul deux le seront dans la phase de saturation). La nature de cette phase dépend de l'application. Couramment cette phase est utilisée comme une notification à toutes les entités pour leur permettre d'arriver à la terminaison locale.

```
PLUG-IN FULL SATURATION .
    • STATUS: S = {AVAILABLE, ACTIVE, PROCESSING, SATURATED};
                SINIT = {AVAILABLE};
    • RESTRICTIONS: R∪T.
AVAILABLE
        SPONTANEOUSLY
            BEGIN
                SEND(ACTIVATE) TO N(x);
                INITIALIZE;
                NEIGHBORS:= N(x);
                IF|NEIGHBORS|=1 THEN
                    PREPARE MESSAGE;
                    PARENT ⇐ NEIGHBORS;
                    SEND(M) TO PARENT;
                    BECOME PROCESSING;
                ELSE BECOME ACTIVE;
                ENDIF
            END
```

```
        RECEIVING(ACTIVATE)
            BEGIN
                SEND(ACTIVATE) TO N(x) − {SENDER};
                INITIALIZE;
                NEIGHBORS:= N(x);
                IF|NEIGHBORS|=1 THEN
                    PREPARE MESSAGE;
                    PARENT ⇐ NEIGHBORS;
                    SEND(M) TO PARENT;
                    BECOME PROCESSING;
                ELSE BECOME ACTIVE;
                ENDIF
            END
ACTIVE
        RECEIVING(M)
            BEGIN
                PROCESS MESSAGE;
                NEIGHBORS:= NEIGHBORS−{SENDER};
                IF|NEIGHBORS|=1 THEN
                    PREPARE MESSAGE;
                    PARENT ⇐ NEIGHBORS;
                    SEND(M) TO PARENT;
                    BECOME PROCESSING;
                ENDIF
            END
PROCESSING
        RECEIVING(M)
            BEGIN
                PROCESS MESSAGE;
                RESOLVE;
            END
```

FIGURE 2.16: La saturation totale

```
PROCEDURE INITIALIZE
BEGIN
    NIL;
END
PROCEDURE PREPARE MESSAGE
BEGIN
    M:=("SATURATION");
END
PROCEDURE PROCESS MESSAGE
BEGIN
    NIL;
END
PROCEDURE RESOLVE
BEGIN
    BECOME SATURATED;
    START RESOLUTION STAGE;
END
```

FIGURE 2.17: Procedures utilisées par la Saturation

Pour déterminer le coût de la saturation en message, il faut remarquer que la phase d'activation n'est qu'un réveil avec k_* initiatrices. Donc $n + k_* - 2$ messages seront échangés. Après le réveil, la phase de détermination des deux nœuds saturés nécessite l'envoi d'un seul message sur chaque liaison sauf la liaison entre les deux nœuds saturés, où il y aura deux messages M. Donc, dans la second phase $n - 1 + 1 = n$ messages sont échangés. Au total :

$$\mathbf{M}[\textit{Full Saturation}] = 2n + k_* - 2$$

Pour déterminer le coût en temps idéal, notons par

$I \subseteq V$ l'ensemble des entités initiatrices,

$L \subseteq V$ l'ensemble des entités feuilles (entités avec un seul voisin) et

$t(x)$ le retard depuis le lancement de l'algorithme jusqu'à ce que l'entité x devienne active.

Pour devenir saturé, un nœud s doit avoir attendu jusqu'à ce que toutes les feuilles deviennent a*ctives* et que le message M qu'elles envoient lui parvienne. Autrement dit, s doit avoir attendu $Max\{t(l) + d(l, s) : l \in L\}$.

Pour devenir actif, un nœud non initiateur x doit avoir attendu le message d'activation (les nœuds initiateurs n'attendent pas pour l'activation). Ainsi

$t(x) = Min\{d(x, y): y \in I\}$ et le delai total vaut :

$$T[\textit{Full Saturation}] = Max\{Min\{d(l, y)\} + d(l, y) : y \in I, l \in L\}$$

2.5.2 Recherche du minimum

La technique de saturation peut être utilisée pour calculer la plus petite valeur parmi un ensemble de valeurs distribuées sur les nœuds d'un arbre. Chaque entité x possède une valeur d'entrée $v(x)$ et toutes les entités sont initialement dans le même statut. Déterminer le minimum parmi ces valeurs signifie que chaque entité doit savoir, à la fin, si oui ou non sa valeur et la plus petite parmi toutes les valeurs de l'ensemble et entrer dans un état *minimum* ou *large*, respectivement. Les valeurs $v(x)$ ne sont pas forcement distinctes ce qui permet à plusieurs entités d'être éventuellement dans l'état *minimum*.

La recherche du minimum dans un arbre avec *racine* (figure 2.18) peut se faire par la stratégie suivante :

- La racine envoie vers le bas (vers les feuilles) la requête pour la recheche du minimum

- A partir des feuilles les nœuds déterminent la valeur minimale et l'envoient vers le haut.
- Les nœuds internes récupèrent l'ensemble des valeurs venant des nœuds qui sont plus bas puis calculent le minimum en fonction des ses valeurs et de leurs propres valeurs puis transmettent le résultat vers les nœuds parents (nœuds plus hauts).
- La valeur minimale sera connue au niveau de la racine

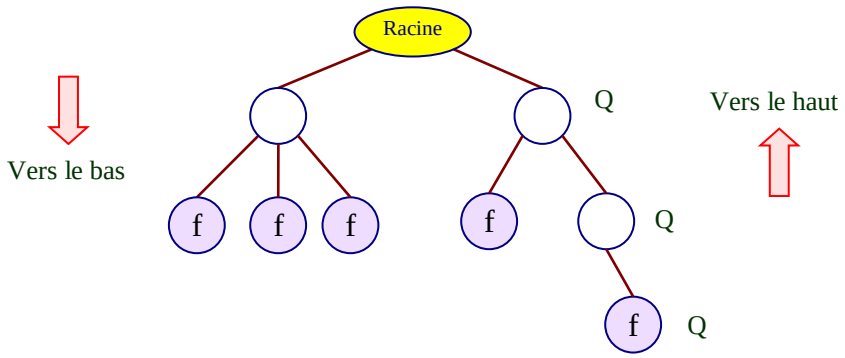

FIGURE 2.18: Arbre avec racine

Avec la technique de saturation totale, on n'a pas besoin d'avoir une racine (ce qui est parfois indeterminable). Il suffit d'inclure, dans le message M, la plus petite valeur connue à l'émetteur. Dans la phase de saturation, les feuilles envoient leurs valeurs avec M et chaque nœud interne envoie le minimum entre sa valeur et celles des récues à son père (le dernier nœud qui ne lui a pas envoyé de message M). La phase de résolution est une notification lancée par les deux nœuds saturés pour communiquer la valeur minimale calculée. De ce fait, n'importe quel nœud détiendra la valeur minimale.

La figure 2.19 donne les changements à introduire dans le protocole de saturation totale pour déterminer le minimum.

```
PROCESSING
        RECEIVING(NOTIFICATION)
        BEGIN
           SEND(NOTIFICATION) TO N(X)−PARENT;
           IF V(X) =RECEIVED VALUE THEN BECOME MINIMUM;
           ELSE BECOME LARGE;
           ENDIF
        END
PROCEDURE INITIALIZE
BEGIN
   MIN:=V(X);
END
PROCEDURE PREPARE MESSAGE
BEGIN
   M:=("SATURATION", MIN);
END
```

```
PROCEDURE PROCESS MESSAGE
BEGIN
   MIN:= MIN{MIN, RECEIVED VALUE};
END
PROCEDURE RESOLVE
BEGIN
   NOTIFICATION:= ("RESOLUTION", MIN);
   SEND(NOTIFICATION) TO N(X)−PARENT;
   IF V(X) =MIN THEN BECOME MINIMUM;
   ELSE BECOME LARGE;
   ENDIF
END
```

FIGURE 2.19: Recerche du minimum avec la technique de saturation

Le nombre de messages transmis pour trouver le minimum sera celui de la saturation totale augmenté du nombre de messages transmis pour la notitification.

$$\boldsymbol{M}[MinF-Tree] = 3n + k* - 4 = (2n + k* - 2) + (n - 2)$$

Le coût en temps sera celui de la saturation augmenté de celui nécessaire à la notification. Si *Sat* denote l'ensemble des deux nœuds saturés, alors :

$$\boldsymbol{T}[MinF-Tree] = \boldsymbol{T}[Full\ Saturation] + Max\{d(s, x) : s \in Sat, x \in V\}$$

2.5.3 Evaluation d'une fonction distribuée

Les fonctions distribuées représentent une classe de problèmes importante. Il s'agit d'accomplir une tâche d'évaluation de fonctions dont les arguments sont répartis sur l'ensemble des nœuds. La recherche du minimum décrite précedamment est un exemple de fonctions distribuées.

Les fonctions qui nous interessent sont les fonctions associatives et commutatives telles que minimum, maximum, somme, produit, etc. ainsi que les predicats logiques. Ces fonctions sont dites opérations semigroupe du fait de leurs propriétés algébriques.

Notons que dans certains cas certains nœuds n'ont pas de valeur, autrement dit $v(x)$ = nil.

Le protocole qui permet le calcul de fonctions sur les arbres est celui de la saturation totale et où les procedures Initialize, Prepare Message, et Process Message sont changées comme indiqué par la figure 2.20. La phase de *resolution* n'est autre qu'une notification de la valeur calculée lancée par les deux nœuds saturés. Nous obtenons cela en modifiant la procédure *Resolve* et en ajoutant la règle qui traite la réception du message de notification.

```
PROCESSING
        RECEIVING(NOTIFICATION)
        BEGIN
            RESULT:= RECEIVED VALUE;
            SEND(NOTIFICATION) TO N(X)−PARENT;
            BECOME DONE;
        END
PROCEDURE INITIALIZE
BEGIN
    IF V(X) ≠ NIL THEN
    RESULT:=F (V(X));
    ELSE
    RESULT:=NIL;
END
PROCEDURE PREPARE MESSAGE
BEGIN
    M:=("SATURATION", RESULT);
END
PROCEDURE PROCESS MESSAGE
BEGIN
    IF RECEIVED VALUE ≠NIL THEN
        IF RESULT ≠NIL THEN RESULT:= F (RESULT, RECEIVED VALUE);
        ELSE RESULT:= F (RECEIVED VALUE); ENDIF
    ENDIF
END
PROCEDURE RESOLVE
BEGIN
    NOTIFICATION:= ("RESOLUTION", RESULT);
    SEND(NOTIFICATION) TO N(X)−PARENT;
    BECOME DONE;
END
```

FIGURE 2.20: Procédures et règle d'évaluation des fonctions distribuées